MEMORIA
PRACTICA 2 - AED2
Problema 9

JOSE RAMON HOYOS BARCELO

Nombre	Grp	e-mail
Angel Ruiz Fernandez	G2.2	a.ruizfernandez@um.es
Alberto Velasco Cotes	G2.2	a.velascocotes@um.es

Cuenta de Mooshak: G2_36

1. Lista de problemas resueltos

ENVIO	PROBLEMA
135	C_AR
55	K_Ba

2. Resolucion de problemas

2.1. Avance rapido

2.1.1. Pseudocodigo y explicación

```
------------------
Algoritmo Principal
------------------
Entrada: número de casos de prueba T
Leer casosPrueba
Para cada casosPrueba:
    Leer n y m
    Leer la matriz de distancias[n][n]

    mejorDiversidad = -1
    mejorSeleccion = vector de n ceros

    Para i = 0 hasta n hacer:
        seleccion = voraz(n, m, distancias, i)
        seleccionados = vector de n ceros
        Para j = 0 hasta n
                Si seleccion[j]
                        añadir j a seleccionar

        diversidad = calcularDiversidad(seleccionados, distancias)

        Si diversidad > mejorDiversidad entonces
            mejorDiversidad = diversidad
            mejorSeleccion = seleccion
        Fin Si
    Fin Para

    Escribir mejorDiversidad
    Escribir mejorSeleccion
Fin Para
```

```
-------------
Avance Rápido
-------------
Función voraz(n, m, distancias, inicio)
    C = conjunto de candidatos
    S = conjunto solución inicial
    enSolucion = vector de n booleanos inicializados en falso
    enSolucion[inicio] = verdadero

    Mientras C no esté vacío y factible(S,m):
        x = seleccionar(S, C, distancias)

        Eliminar x de C
        Si factible(S,m)
            Añadir x a S
            enSolucion[x] = verdadero
        Fin Si
    Fin Mientras

    resultado = vector binario de tamaño n
    Para i = 0 hasta n hacer:
        Si enSolucion[i] entonces resultado[i] = 1
        Si no, resultado[i] = 0
    Fin Para

    Devolver resultado
Fin Función

-----------
Seleccionar
-----------
Función seleccionar(S, C, distancias)
    mejorGanancia = -1
    mejorCandidato = -1

    Para cada candidato en C hacer:
        ganancia = 0
        Para cada s en S hacer:
            ganancia = ganancia + distancias[candidato][s] +
                distancias[s][candidato]
        Fin Para

        Si ganancia > mejorGanancia entonces
            mejorGanancia = ganancia
            mejorCandidato = candidato
        Fin Si
    Fin Para

    Devolver mejorCandidato
Fin Función

--------
Factible
--------
Funcion factible(S,m)
    Devolver (tamaño de S < m)
```

Este algoritmo se basa en 4 funciones:

- La función ’calcularDiversidad’ suma la distancia entre cada par de elementos del subconjunto S. Para ello se usa un doble bucle para recorrer todos los pares. En el segundo bloque se empieza desde i para evitar contar dos veces la misma pareja de elementos, ya que sumamos ambas direcciones.

- La función ’seleccionar’ selecciona el candidato en C que tenga mayor ganancia. Se usa un doble bucle para recorrer cada candidato y, sobre cada uno, recorrer S el cual contiene las soluciones.

- La función ’factible’ la cual devuelve verdadero si todavía hay elementos por seleccionar y falso si se han alcanzado el número de elementos.

- La función ’voraz’ la cual inicia la solución S con un elemento de partida (en este caso el parámetro ’inicio’). Mientras que queden candidatos en C y sea factible (todavía hay elementos por seleccionar), va seleccionando los candidatos de mayor ganancia, los elimina de S y, si es factible, se añade en S. Usamos un vector de booleanos, el cual contiene verdadero en i posición si este forma parte de la solución. Como el ejercicio pide que sea un vector de 0’s y 1’s, lo convertimos.

’C’ es el conjunto de candidatos disponibles, el cual se va actualizando eliminando el candidato elegido (así nos aseguramos de que no se selecciona dos veces).

’S’ es el conjunto solución, el cual contiene la solución actual y crece hasta alcanzar el tamaño m.

Como la solución final depende del elemento inicial escogido, recorremos cada posible inicio y seleccionamos la solución con mayor diversidad.

2.1.2. Programación del algoritmo

```
#include <iostream>
#include <vector>
using namespace std;

/*
 *Calcula la diversidad de un subconjunto
 *La función recorre todos los pares de elementos del vector ’seleccionados’ y
 *acumula la suma de sus distancias.
 *Devuelve un entero, que sería el total de las distancias.
*/

int calcularDiversidad(const vector<int>& seleccionados, int** distancias) {
    int suma = 0; //entero que representa la diversidad total
    for (size_t i = 0; i < seleccionados.size(); ++i) {
        //Se suman ambas direcciones de la distancia
        for (size_t j = i + 1; j < seleccionados.size(); ++j) {
            suma += distancias[seleccionados[i]][seleccionados[j]] +
                distancias[seleccionados[j]][seleccionados[i]];
        }
    }
    return suma;
}
```

```
/*
 *Elige el candidato con más ganancia
 *Dado el conjunto soluciones S y el conjunto de candidatos C, la función
 *determina que cancidato es
 *mejor sumando la ganancia de añadirlo a S
 *Devuelve el mejor candidato.
*/

int seleccionar(const vector<int>& S, const vector<int>& C, int** distancias) {
    int mejorCandidato = -1; //entero que representa el mejor candidato
    int mejorGanancia = -1; //entero que representa la mejor ganancia

    for (int candidato : C) {
        int ganancia = 0; //ganancia del candidato seleccionado
        for (int s : S) {
            ganancia += distancias[candidato][s] + distancias[s][candidato];
        }

        if (ganancia > mejorGanancia) {
            mejorGanancia = ganancia;
            mejorCandidato = candidato;
        }
    }

    return mejorCandidato;
}

/*
 *Determina si es factible seguir agregando elementos
 *La función devuelve verdadero si el numero de elementos de S es menor que m y
 *falso si es mayor
*/
// Función factible: true si no se ha alcanzado el tamaño m
bool factible(const vector<int>& S, int m) {
    return S.size() < (size_t)m;
}
```

```
/*Función Avance rápido
 *A partir de un elemento de inicio, se construye la solución S agregando en
 *cada iteración el mejor candidato
 *n es el número total de elementos y m el número de elementos a seleccionar
 *'distancias' contiene las distancias entre cada par de elementos
 *'inicio' es el elemento inicial
*/
vector<int> voraz(int n, int m, int** distancias, int inicio) {
    vector<int> C; // conjunto de candidatos
    for (int i = 0; i < n; ++i) {
        if (i != inicio)
            C.push_back(i);
    }

    vector<int> S = {inicio}; // solución inicial con el elemento de inicio
    vector<bool> enSolucion(n, false); //vector booleano
    enSolucion[inicio] = true;

    while (!C.empty() && factible(S, m)) {
        int x = seleccionar(S, C, distancias); // seleccionar candidato

        // Construir nuevo conjunto de candidatos sin x
        vector<int> nuevoC;
        for (int c : C) {
            if (c != x) nuevoC.push_back(c);
        }
        C = nuevoC;

        // Insertar si es factible
        if (factible(S, m)) {
            S.push_back(x);
            enSolucion[x] = true;
        }
    }

    // Convertir solución a vector de 0 y 1
    vector<int> resultado(n);
    for (int i = 0; i < n; ++i)
        resultado[i] = enSolucion[i] ? 1 : 0;
    return resultado;
}
```

```
int main() {
    int T;
    cin >> T;
    while (T--) {
        int n, m;
        cin >> n >> m;

        // Reservar espacio para la matriz de distancias
        int** distancias = new int*[n];
        for (int i = 0; i < n; ++i)
            distancias[i] = new int[n];

        //Lectura de matriz
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                cin >> distancias[i][j];

        int mejorDiversidad = -1;
        vector<int> mejorSeleccion;

        //evaluar el algoritmo para cada posible elemento de inicio
        for (int i = 0; i < n; ++i) {
            vector<int> seleccion = voraz(n, m, distancias, i);

            //extraer índices de elementos seleccionados
            vector<int> seleccionados;
            for (int j = 0; j < n; ++j)
                if (seleccion[j]) seleccionados.push_back(j);

            //calcular diversiad
            int diversidad = calcularDiversidad(seleccionados, distancias);

            //seleccionar mejor solución
            if (diversidad > mejorDiversidad) {
                mejorDiversidad = diversidad;
                mejorSeleccion = seleccion;
            }
        }

        //imprimir resultado
        cout << mejorDiversidad << endl;
        for (int bit : mejorSeleccion)
            cout << bit << " ";
        cout << endl;

        //liberar memoria
        for (int i = 0; i < n; ++i)
            delete[] distancias[i];
        delete[] distancias;
    }

    return 0;
}
```

2.1.3. Estudio teórico

- Para la función 'calcularDiversidad' se recorren los pares distintos(i,j). Como el tamaño máximo es m, tendría O(m²).

- Para la función 'voraz' necesitamos tener en cuenta la función 'seleccionar'. Esta recorre todos los candidatos (en el peor de los casos hasta n-1), y para cada uno calcula su ganancia junto con los elementos acutales de la solución (como máximo hasta m-1). Por lo tanto el orden de la función 'seleccionar' es O(n*m).

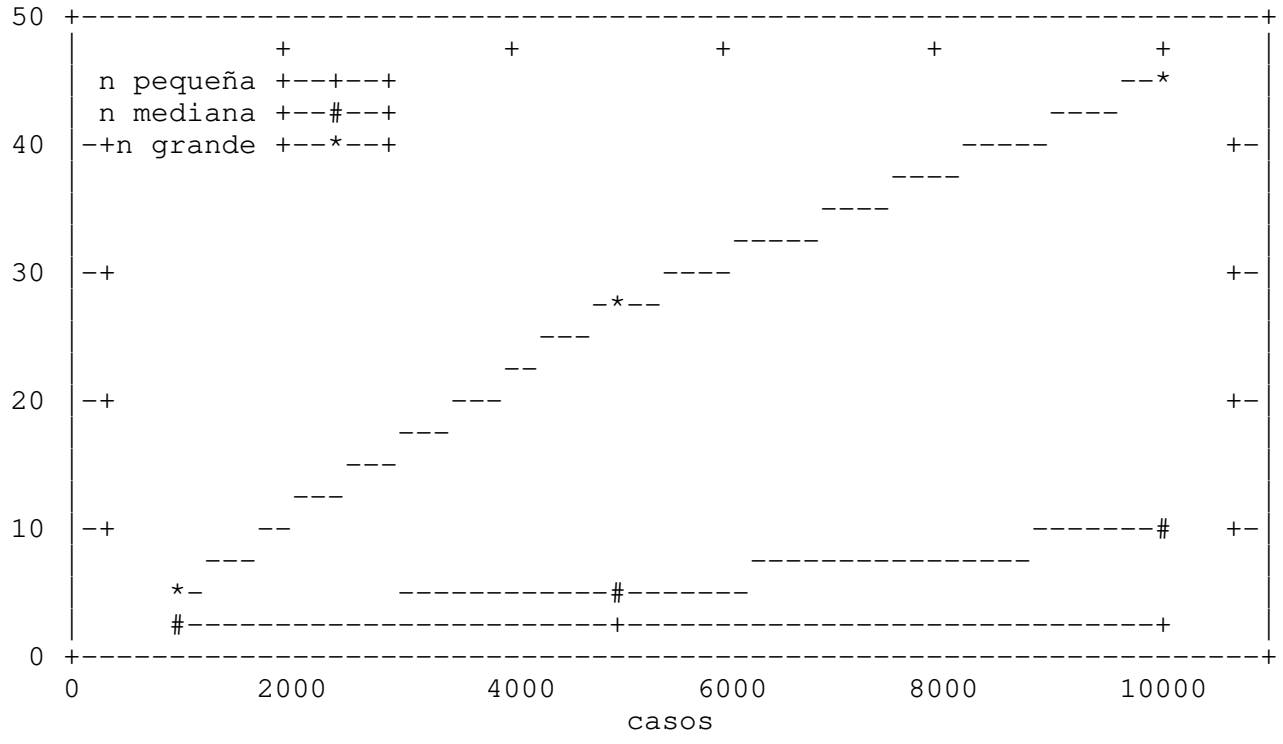
La función 'voraz' hace eso para cada candidato. Por lo tanto, el orden de la función sería O(n*m²) para el peor de los casos (cuando se realizan todas las iteraciones posibles).
Para el mejor caso (número de candidatos pequeño o m pequeño), el orden sería de O(n*m).

2.1.4. Estudio experimetal

Para realizar este estudio experimental, hemos generado diferentes matrices con distinto tamaño de n y m (hemos decidido que m = n/2).

- Primero hemos generado matrices con un n entre 5 y 15.
- Después hemos generado matrices con un n entre 15 y 30.
- Por último hemos generado matrices con un n entre 30 y 50.

Para cada una, hemos probado con diferente número de casos. En concreto, con 1000 , 5000 y 10000 casos.
Con todas estas pruebas hemos creado una gráfica la cual representa el tamaño de las entradas en el eje X y los tiempos en el eje Y.



2.1.5. Contraste de estudio teorico y experimental

En la gráfica se puede ver que las entradas con distancias más grandes aumentan significativamente el tiempo de ejecución, mientras que las distancias pequeñas muestran tiempos de ejecución más bajos.

Podemos apreciar que no se aprecian discrepancias entre el estudio teórico y el experimental, por lo tanto, coinciden en ambos casos. Ambos muestran que el algoritmo es eficiente para valores pequeños de n, y que según va aumentando, el algoritmo empieza a ser más costoso.

El tiempo de ejecución obtenido se debe principalmente al crecimiento cuadrático en el cálculo de la diversidad y selección de elementos. También se debe a la evaluación de cada elemento candidato con todos los elementos seleccionados actuales. Esto hace que el tiempo de ejecución del algoritmo aumente significativamente.

2.2. Backtracking

2.2.1. Pseudocódigo y explicación

Esquema de backtracking de optimización base, con variables para representar el peso repartido a cada equipo, y el par mas optimizado, usando la diferencia entre ellos como vao (valor optimo actual).

La solución se recorre usando un vector ‘sol‘ que por cada peso del vector dado indica a que equipo va, A o B.

Dentro del bucle, antes de generar un hermano se deshace el anterior ya que usamos las mismas variables acumulativas. El hermano se genera incrementando la combinación del vector solucion y aplicando la solución a los pesos de los equipos (cumulativo).

Si es solución (se usan todos los pesos y los equipos se diferencian en no más de 1), se compara con la solución optima anterior y se asigna si es más optimo. Si no es solución, se comprueba si esta rama puede dar soluciones, en cuyo caso se profundiza (busqueda primero en profundidad) y si no se descarta el subarbol.

Se puede profundizar si esta por debajo del nivel final, y los pesos restantes son mayores que la diferencia actual, y la diferencia en numero entre equipos es menor que los nieveles restantes.

Retroceder consiste en deshacer la sumación del peso escogido de las variables cumulativas, desdefinir la solucion en el nivel actual y volver al nivel anterior.

Concluido el bucle, asumiendo que se ha encontrado solución, se asegura de que el equipo A sea mas pequeño de B para resolver la indeterminación del orden, y se retorna la solución compuesta de los pesos de cada equipo.

```
Entero, Entero BT(pesos):
    Entero nivel, voa # valor optimo actual

    Entero pesoActA, pesoActB   # peso de equipos actual
    Entero pesoOptA, pesoOptB   # peso equipos optimo (solucion)

    Vector s(pesos.size)        # vector solución (cada peso 0 = A, 1 = B)

    mientras nivel >= 0:        # bucle iterativo
        # deshacer hermano anterior para generar otro
        si s[nivel] = 0:
            pesoActA = -pesoActA[nivel]
        sino s[nivel] = 1:
            pesoActB = -pesoActB[nivel]

        # generar hermano
        s[nivel]++
        # computar pesos sumando el peso actual al equipo correspondiente
        si s[nivel] = 0:
            pesoActA = pesoActA + pesos[nivel]
        sino si s[nivel] = 1:
            pesoActB = pesoActB + pesos[nivel]

        # diferencia entre equipos
        Entero diferencia = |pesoActA - pesoActB|

        # si es solución
        si nivel == n - 1:          # se usan todos los pesos
            si diferencia < voa:    # si es mas optimo, cambiar
                voa = diferencia
                pesoOptA = pesoActA
                pesoOptB = pesoActB

        # criterio de poda
        si
          nivel != n - 1
          && sum(pesos[nivel:]) > diff                          # criterio 1
          && |contar(s, 0) - contar(s, 1)| < n - nivel:         # criterio 2
            nivel = nivel + 1
        # retroceder
        sino:
            mientras no s[nivel] < 1 and nivel >= 0:  # mientras queden herm.
                # deshacer
                si s[nivel] == 0:
                    pesoActA = pesoActA - pesos[nivel]
                sino si s[nivel] == 1:
                    pesoActB = pesoActB - pesos[nivel]
                s[nivel] = -1
                nivel = nivel - 1

    # solución, si son diferentes, el equipo A sea el mas ligero
    si pesoOptA > pesoOptB:
        intercambiar(pesoOptA, pesoOptB)

    retornar pesoOptA, pesoOptB
```

2.2.2. Programación del algoritmo

```
std::pair<int, int> bt(std::vector<int> pesos) {
    auto type = decltype(pesos)::value_type(0); // tipo a acumular
    int n = pesos.size();                       // tamaño de vectores
    // ordenar pesos: asignar pesos mas significativos primero
    std::sort(pesos.begin(), pesos.end(), std::greater<int>());

    int nivel = 0, voa = INT_MAX;   // nivel y valor optimo actual

    // por equipo
    int pesoActA = 0, pesoActB = 0; // variables cumulativas solucion actual
    int pesoOptA = 0, pesoOptB = 0; // variables optimas

    std::vector<int> s(n, -1);      // vector solucion

    while (nivel >= 0) {
        // deshacer hermano anterior y generar
        if (s[nivel] == 0)
            pesoActA -= pesos[nivel];
        s[nivel]++;
        if (s[nivel] == 0)
            pesoActA += pesos[nivel];   // sumar peso al equipo correspondiente
        else if (s[nivel] == 1)
            pesoActB += pesos[nivel];

        int diff = std::abs(pesoActA - pesoActB);   // computar diferencia
        // en solucion
        if ((nivel == n - 1) && (diff < voa)) {
            voa = diff;
            pesoOptA = pesoActA;
            pesoOptB = pesoActB;
        }
        // criterio de poda para profundizar
        if (
            (nivel != n - 1)
            && (std::accumulate(pesos.begin() + nivel, pesos.end(),type) > diff)
            && (std::abs(std::count(s.begin(), s.end(), 0)
                - std::count(s.begin(), s.end(), 1)) < n - nivel)
        )
            nivel++;
        // retroceder
        else
            while (!(s[nivel] < 1) && nivel >= 0) {
                // deshacer nivel
                if (s[nivel] == 0)
                    pesoActA -= pesos[nivel];
                else if (s[nivel] == 1)
                    pesoActB -= pesos[nivel];
                s[nivel] = -1;
                nivel--;
            }
    }

    // solucion ordenada
    if (pesoOptA > pesoOptB)
        std::swap(pesoOptA, pesoOptB);

    return std::pair<int, int>(pesoOptA, pesoOptB);
}
```

2.2.3. Estudio teórico

Cada peso puede estar en uno u otro equipo, indicando que la forma de backtracking es de arbol binario. El tamaño de los equipos solo puede diferir en una persona. Por tanto, mediante combinatoria habrá como maximo

$$s(n) = \binom{n}{n/2} + \binom{n}{n/2-1} + \binom{n}{n/2+1} \quad (1)$$

soluciones con n pesos. Esto se puede entender como que hay n C n/2 formas de escoger la mitad de pesos del vector pesos (que van a un equipo, y la otra mitad al otro equipo), más formas de legir n/2−1 y n/2+1 elementos.

Usando esta función y sabiendo que el arbol es binario, se puede inferir que va a tener

$$a(n) = 2^{(\log_2(s(n))+1)} - 2 \quad (2)$$

nodos, en el peor caso, sin contar el raiz. Ya que el numero de nodos de un arbol binario es 2^(h+1) − 1 siendo h la altura del arbol, y la altura de un arbol binario es el log2(k) siendo k el numero de nodos hoja (asumiendo que está completamente balanceado). Como observación, el numero de nodos suele ser de un orden de magnitud mas que el numero de soluciones potenciales.

Como es normal en backtracking, este algoritmo es O(2^logn), bastante grande. Además en la implementación se ordena el vector de pesos tal que se asignen los pesos mas grandes primero, y los ajustes pequeños después, que tiene una complejidad de O(nlogn) en el caso promedio, pero comparado con la complejidad del backtracking, el tiempo de ordenación es despreciable. Así la ordenación permite la aplicación eficiente de las condiciones de poda, pero estos son muy dificiles de formular matematicamente. Se verán en el estudio experimental.

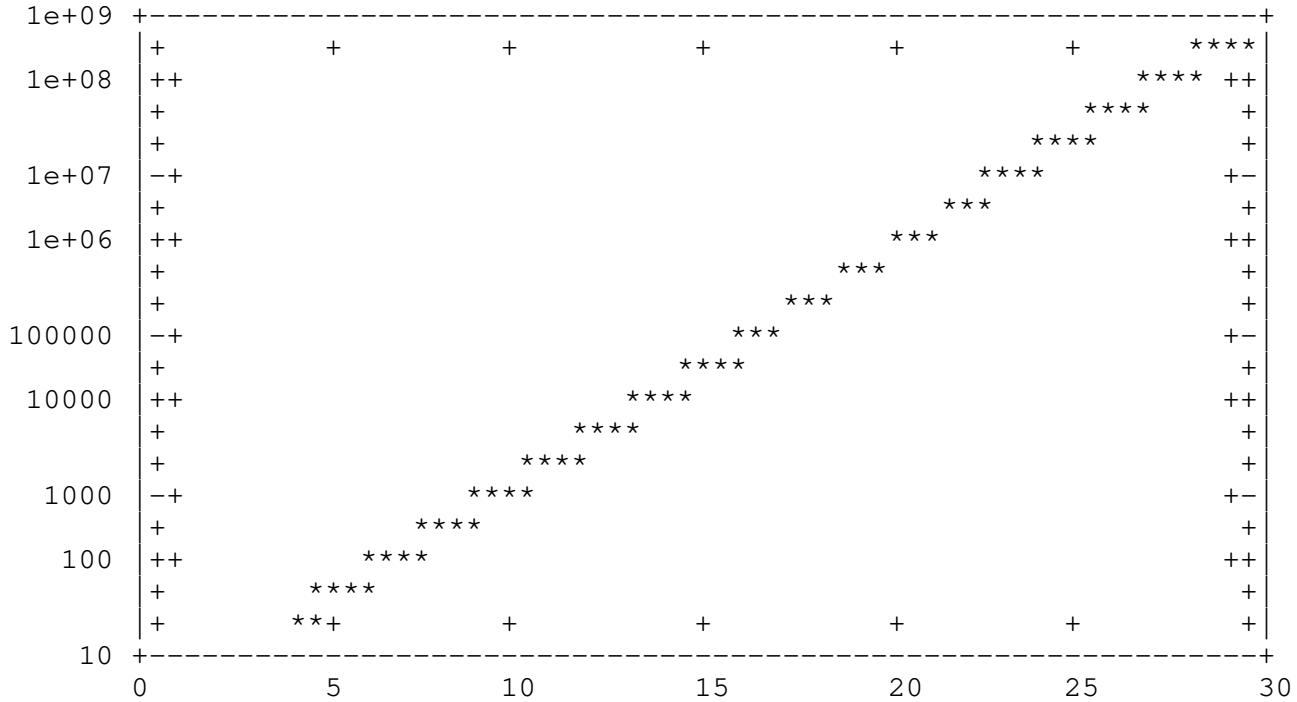


Figura 1. Grafico logaritmo del crecimiento del tamaño del arbol, lineal porque el tamaño crece de forma exponencial.

2.2.4. Estudio experimetal

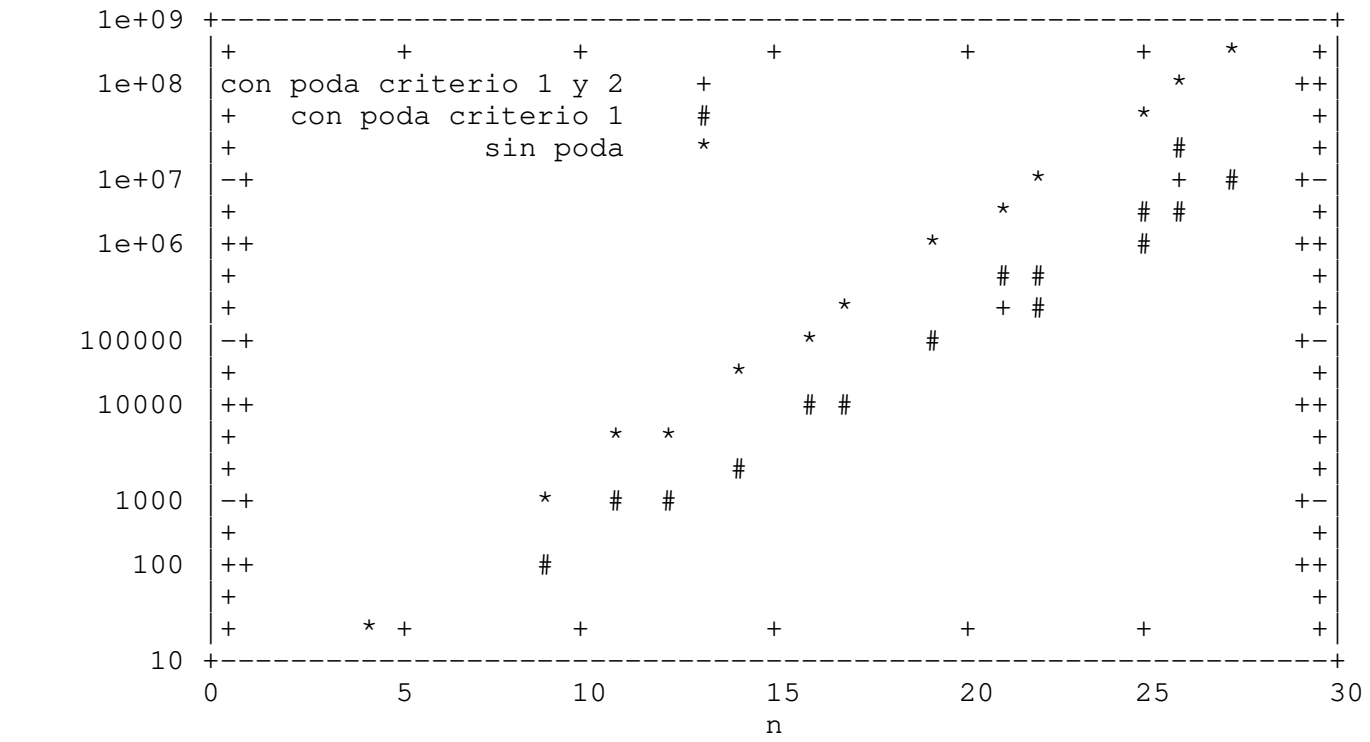


Figura 2. Grafico logaritmico de tamaño de vector pesos vs tamaño del arbol con y sin poda.

poda	A	B	R^2
con 1,2	0.23569	0.58473	0.9925
con 1	0.20940	0.59397	0.9928
sin	0.6258911	0.6961438	1

Tabla 1. Regresión exponencial del tamaño de los arboles en forma y = A*e^(B*x)

El tamaño del arbol es efectivamente exponencial. El criterio 1 es el que hace la mayoría de la poda, mientras que el criterio 2 influye menos.

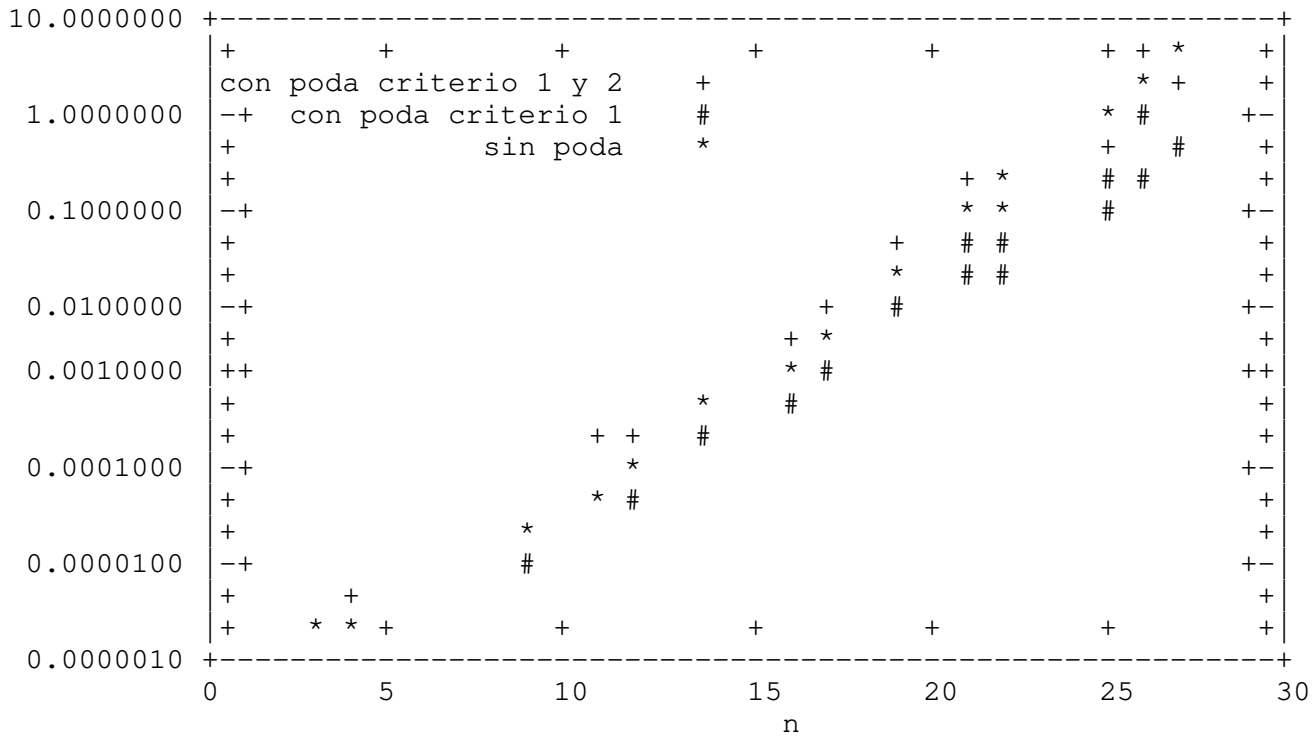


Figura 3. Grafico de tiempo por prueba vs tamaño de vector de pesos con y sin poda.

poda	A	B	R^2
con 1,2	2.29e-07	1.8348	0.9914
con 1	1.52e-07	1.743509	0.9885
sin	1.01e-07	1.891884	0.9944

Tabla 2. Regresión exponencial de los tiempos

En cuanto al tiempo, sin poda tima mas tiempo, con el criterio 1 se rebaja bastante el tiempo, sin embargo con los dos criterios, el tiempo aumenta casi hasta alcanzar el tiempo sin poda.

2.2.5. Contraste de estudio teorico y experimental

Los resultados teoricos y experimentales concuerdan sin poda y con el criterio de poda 1. Sin embargo, cuando se activa el criterio 2, el tiempo aumenta en vez de reducirse, ¿por qué?

Los criterios no tienen coste cero. El criterio 1 tiene un bucle que calcula la suma de elementos restantes, y el criterio 2 tiene bucles que calculan a que equipo pertenece cada peso, que se ejecutaría n veces por cada nodo del arbol, haciendo un orden O(a(n)*n) ~= O(n*2^n).

El primer criterio poda mucho mas de los ciclos que cuesta, por eso reduce significativamente el tiempo. Pero el segundo, reduce poco la poda costando bastante mas ciclos, con lo cual el tiempo sube bastante.

3. Conclusiones

3.1. Avance rapido

El avance rapido es una buena forma de encontrar soluciones posibles rapidamente sin tener las restricciones de ser optima o unica en problemas np-completos ya que es mucho mas rapida (de ahi el nombre) y eficiente para lo buena que es.

3.2. Backtracking

A no ser que se pueda podar muchisimo el arbol, el backtracking es extremadamente costoso computacionalmente, inalcanzable para problemas grandes.

Sin embargo, con problemas np-completos, es mejor que el metodo ingenuo para encontrar todas las soluciones posibles o la solución mas optima. Sin embargo, si solo se requiere de una solución suboptima, algoritmos como el avance rapido o los de programación dinamica son muchisimo mas eficientes.

3.3. Conclusión

En conclusión, esta práctica nos ha ayudado sobre todo a poder comprender el tema con más facilidad ya que, el hecho de poder programar uno mismo el algoritmo nos hace tener en cuenta ciertos factores que quizá no hubiéramos tenido en cuenta solo con la teoría.
Esto es clave para poder entender como los algoritmos se comportan en situaciones reales